In Situ Magnetic Flux Vortex Visualization in Time-Dependent Ginzburg-Landau Superconductor Simulations

Hanqi Guo [1*] Tom Peterka [1†] Andreas Glatz [2‡]

1) Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
2) Materials Science Division, Argonne National Laboratory, Lemont, IL, USA

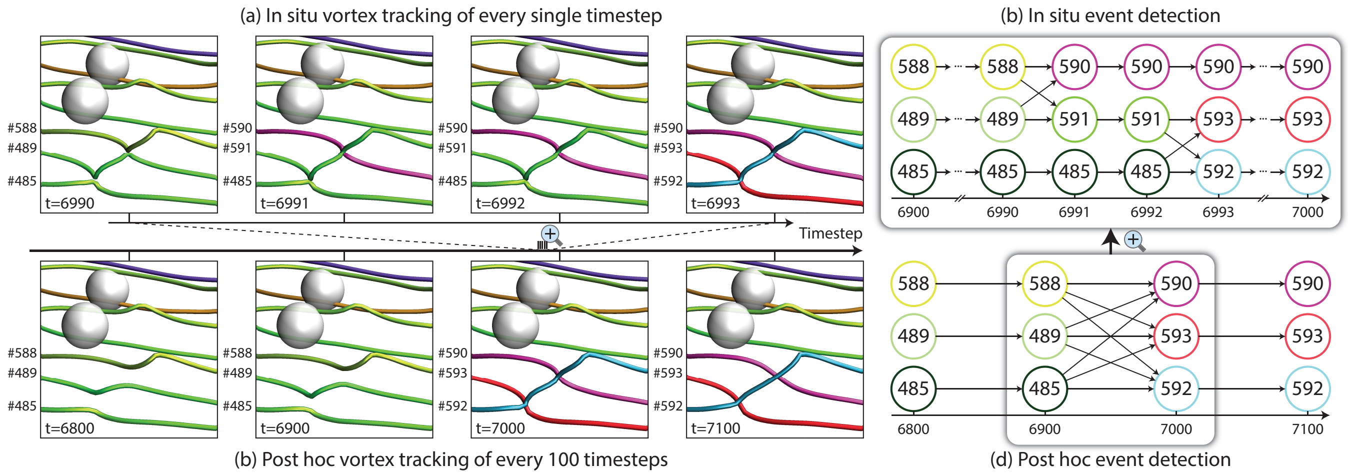

(a) In situ vortex tracking of every single timestep
(b) In situ event detection
#588
#489
#485
#590
#591
#593
#592
t=6990
t=6991
t=6992
t=6993
Timestep
t=6800
t=6900
t=7000
t=7100
588
489
485
590
591
593
592
6900
6990
6991
6992
6993
7000
6800
7100
(b) Post hoc vortex tracking of every 100 timesteps
(d) Post hoc event detection

Figure 1: Comparison between in situ (a and b) and post hoc (c and d) visualization of magnetic flux vorticies in superconductors. The in situ method processes every single timestep during the simulation, and the post hoc processes every 100 timesteps after the simulation. The post hoc analysis show that three vortices #598, #489, and #485 recombine with each other and formed new three vortices #590, #593, and #592 within the interval of 100 timesteps (from 6900 to 7000). The in situ visualization precisely shows that #588 and #489 first recombine into #590 and #591 at timestep 6990, and then #591 and #485 recombine into #593 and #592 at timestep 6992. The semi transparent spheres are material defects that attract vortices.

Abstract

We present an in situ visualization framework to capture comprehensive details of vortex dynamics in superconductor simulations. Vortices, which determine all electromagnetic properties of type-II superconductors, are extracted and tracked at the same time with GPU-based time-dependent Ginzburg-Landau superconductor simulations. The in situ workflow involves three parts: 1) a tightly-coupled GPU-accelerated algorithm that detects primitives for ambiguity-free vortex tracking, 2) a loosely-coupled task-parallel feature-tracking method, and 3) a web-based remote visualization tool for vortex dynamics analysis. Our design minimizes the data movement and storage, maximizes the resource utilization, and reduces the slowdown of the simulation. Our solution captures all vortex dynamics in the simulation, previously impossible with traditional post hoc methods. We also demonstrate in situ visualization cases that help scientists understand how vortices cut each other and recombine into new vortices, which are directly related to energy dissipation of superconducting materials.

Keywords: In situ visualization, superconductivity, feature extraction and tracking, GPU algorithm, web-based visualization.

*hguo@anl.gov
†tpeterka@mcs.anl.gov
‡glatz@anl.gov

1 Introduction

High temperature (type-II) superconductors, which can conduct current with zero electrical resistance at liquid nitrogen temperature, are widely used in power transmission, particle accelerators, and magnetic resonance imaging. A key phenomenon emerging in those systems is *magnetic flux vortices*, or simply vortices, which determine the energy dissipation and all electromagnetic responses in type-II superconductors. To design superconductors that can sustain higher electric currents for applications, materials scientists must understand and control the behavior of vortices.

An effective way to understand vortex behavior is to simulate the superconductors by using large-scale numerical solvers. Here we consider simulations based on the Ginzburg-Landau theory of superconductivity, which models the superconducting properties in terms of a complex-valued *order parameter* field $\psi \in \mathbb{C}$. In this theory, vortices are defined as the singularities in ψ, which are 3D curves that satisfy

$$|\psi| = 0 \text{ and } -\oint_C \nabla\theta \cdot d\mathbf{l} = 2n\pi, \tag{1}$$

where θ and $|\psi|$ are the phase angle and magnitude of ψ, respectively, C is a small closed contour that encircles the singularity, and n is usually ± 1 and indicates the chirality of the vortex with respect to C. Notice that the definition of magnetic flux vortices is related to but different from that of fluid flow vortices. Vortices in superconductors are 3D curves, which are comparable to vortex core lines or swirling centers in fluid flows. In the rest of this paper, unless otherwise noted, the term vortex means magnetic flux vortex.

The motivating scientific questions in this paper are when and how exactly vortices recombine with each other [12], specifically, the accurate time of the cuttings and the behaviors of vortices before and after such events. These questions require vortex tracking in high temporal resolution, because vortices move rapidly around events. Unfortunately, in current post hoc visualization workflow, scientists have to dramatically reduce the data written to the storage system because of prohibitive I/O bandwidth and capacity. In practice, $10^2 - 10^4$ timesteps are skipped in the simulation, in order not to slow the simulation and overflow the storage.

We extract and track vortices *in situ*—as opposed to traditional post hoc processing—to address the precision problem and reduce the I/O demand. As shown in Figure 1, the temporal resolution in traditional post hoc processing is usually insufficient to capture dynamics of interest in the vicinity of topological changes (so-called events). Such a problem can be solved only by accessing all timesteps during the simulation. As an added benefit, the I/O demand of writing vortices and analysis results, instead of raw simulation output, is dramatically reduced.

The entire in situ vortex visualization workflow consists of two parts—online processing and post hoc visualization. The online processing extracts, tracks, analyzes, and reduces vortices in situ with the simulation; the post hoc visualization provides an interactive user interface to explore and analyze the online processing results.

We design and implement the online processing procedure for the time-dependent Ginzburg-Landau (TDGL) simulations based on the characteristics of both simulation and vortex tracking algorithms. The simulation code, *GLGPU*, is a large-scale GPU-based partial differential equation (PDE) solver [24]. The output order parameter field is available in GPU memory after every iteration of the solver. The vortex tracking algorithm, which has already been used in post hoc analysis, consists of two stages—detecting primitives and transforming the primitives into vortices. The two stages are based on numerical analysis and graph traversal, respectively. In the first stage, we check whether every mesh face or edge is intersected with a vortex, because the intersections indicate the locations and movements of vortices. In the second stage, we transform the primitives into vortices and their trajectories over time that are modeled with connected graphs.

The major challenge in our application is that the GPU-based simulation runs much faster than the analysis algorithms do. We must design faster parallel and asynchronous algorithms, in order to minimize the slowdown of the simulation. The online processing hence consists of two components—the tightly-coupled, synchronous, GPU-accelerated *primitive detector* and the loosely-coupled, asynchronous, task-parallel *vortex analyzer*. The two components play the role of producer and consumer, respectively. The producer runs at every single iteration of the GPU-based simulation, without any data movement of the order parameter field. The primitives are then transferred to the consumer for vortex reconstruction and analysis. We run the consumer asynchronously as an independent process, in order to avoid slowing the simulation with the time-consuming graph analysis on CPUs. Furthermore, we accelerate the consumer with a novel task-parallel approach, because the vortex analyzer usually runs slower than the simulation and the primitive detector. The output analysis results are written to a database for post hoc visualization. In the performance benchmarks, the amount of output is only $10^3 - 10^6$ smaller than the original order parameter data.

The post hoc visualization enables users to explore and further analyze vortex dynamics that are drawn from the simulation. The tool not only contains traditional 3D visualizations but also provides 2D visualizations of events and distances between vortices, which help users explore when vortices recombine and how they move before and after the events. The web-based tool provides a remote visualization solution and works flexibly on workstations or mobile devices without installing any specific software.

Compared with previous studies on vortex tracking in superconductors [13, 21], our in situ solution enables users to discover more details in the simulations given the fine temporal resolutions. We also improve the performance of vortex tracking in two ways. First, we develop a GPU-based vortex primitive detection algorithm that can run 200 times faster than the CPU implementation. Second, we parallelize the graph-based vortex reconstruction algorithm, which can make full use of all available CPU cores on the node. In summary, the contributions of this paper are fourfold:

- An in situ vortex visualization framework for analyzing and understanding vortex dynamics in superconductors
- A GPU-accelerated algorithm to detect primitives for ambiguity-free vortex extraction and tracking
- A task-parallel feature-tracking method that extracts and tracks vortices asynchronously
- Applications of in situ analysis of vortex dynamics and macro-behaviors of superconductors

The remainder of this paper is organized as follows. Section 2 reviews the background of in situ visualization, vortex visualization, and TDGL simulations. Section 3 presents our in situ workflow, and Sections 4 and 5 detail the primitive detector and the vortex analyzer, respectively. Section 6 describes the post hoc visualization. Implementation details and performance benchmarks are in Sections 7 and 8, respectively. Section 9 demonstrates applications cases, followed by the conclusions in Section 10.

2 Background

We review the background in vortex extraction and tracking, in situ visualization, and TDGL simulations.

2.1 Vortex extraction and tracking

We distinguish the concepts of fluid flow vortices from magnetic flux vortices, which are fundamentally different. In fluid dynamics, vortices can be defined in various ways, such as λ_2 [14, 26], to characterize swirling motions. Comprehensive literature reviews on fluid flow vortices are available in [15] and [23]. In general, fluid flow vortices are localized in two ways—regions and core lines. In the former, vortex regions are usually extracted by thresholding the derived scalar field such as λ_2. In the latter, a vortex core line is defined by a locus of points that satisfy certain criterion, such as extrema of vorticity magnitude. The parallel vectors operator [20] and the feature flow fields [27], respectively, are the established techniques to extract and track vortices in fluid flows.

In the Ginzburg-Landau theory, vortices are well defined as topological defects in a complex-valued order parameter field. The vortex extraction algorithm is based on this theorem in complex scalar fields: there must be an equal number of vortex entry and exit points for any given closed volume, such as a mesh cell in the simulation. As shown in Figure 2(a), one can first localize the punctured points, that is, entries and exits on all mesh faces with the line integral (Equation 1), and then connect the punctured points based on the graph that describes cell/face neighborships in the mesh [22]. The vortex tracking algorithm extends the vortex extraction into 4D (space and time), based on the fact that the above theorem still holds in 4D [13, 21]. As shown in Figure 2(b), the movement of a punctured point can be captured by detecting punctured *virtual* faces of the prism, that is, the space-time extrusion of the mesh face. Vortices and their moving trajectories can then be constructed based on the graph analysis (Figure 2(c)).

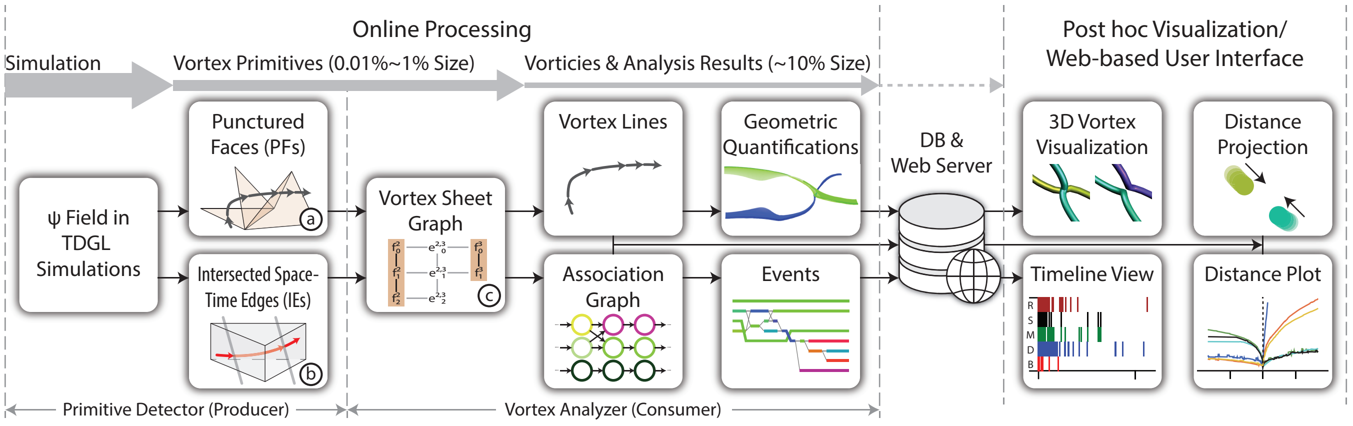


Figure 2: Our in situ vortex visualization workflow, which consists of the online processing and the post hoc visualization. The online processing extracts, tracks, analyzes, and reduces vortices during the simulation; and the post hoc visualization provides an interactive user interface to further explore and analyze vortices.

We review the two-stage algorithm of the magnetic flux vortex extraction and tracking: primitive detection and graph analysis. In the primitive detection stage, for two timesteps i and j, we detect *punctured faces* (PFs) $\mathcal{F}_i$ and space-time *intersected edges* (IEs) $\mathcal{E}_i^j$. In the graph analysis stage, we construct vortices in two individual timesteps ($\{\mathcal{V}_i^k\}$ and $\{\mathcal{V}_j^{k\prime}\}$) based on $\mathcal{F}_i$ and $\mathcal{F}_j$, respectively. We then build the *association graph* $\mathcal{A}_i^j(k, k\prime)$ based on $\mathcal{F}_i$, $\mathcal{F}_j$, and $\mathcal{E}_i^j$. The nodes of the association matrix are $\{\mathcal{V}_i^k\}$ and $\{\mathcal{V}_j^{k\prime}\}$, and the link $(k, k\prime)$ indicates that two vortices $\mathcal{V}_i^k$ and $\mathcal{V}_j^{k\prime}$ are associated. An *event* happens if a vortex has more than one link.

Compared with previous studies, in this paper we develop a GPU-accelerated primitive detection algorithm and a task-parallel graph analysis for vortex analysis, in order to enable efficient in situ processing with the simulation. We also propose a method to remove ambiguity cases in previous studies (see discussions in Section 4).

2.2 In situ visualization

As summarized by a recent comprehensive review [2], in situ visualization is motivated by multiple factors—the disparity between scientific computing and I/O rate, the increased temporal resolution needed for accurate data analysis, and the utilization of all computing resources. Our in situ workflow benefits the analysis of vortex dynamics in all three aspects. We categorize the related work on in situ visualization into applications, algorithms, and infrastructures.

In situ visualization techniques have been used in various scientific applications. For example, Yu et al. [33] visualized volume and particle data in combustion simulations. Topologies in combustion data can be extracted by using segmented merge trees in situ [16]. Woodring et al. [31] developed an in situ workflow to analyze eddies in the study of ocean-climate models. Fabian [10] used in situ processing to detect fragments in explosion simulations. All these studies aim to achieve higher temporal resolution of the data in order to extract time-critical features. In our application, an additional challenge is that the TDGL simulation runs much faster than the vortex analysis algorithms. We must redesign the workflow and accelerate the algorithms in our in situ processing.

Various algorithms are used for efficient in situ visualization. For example, the performance of in situ volume rendering depends on image compositing algorithms, such as binary swap [18] and 2-3 swap [34]. Explorable images [28], which can generate new volume rendering results with a small number of rendered images, can be used in in situ visualization. Similarly, pathtubes can be visualized in situ with explorable images [32]. Ahrens et al. presented an image-based in situ visualization framework for interactive exploration [1]. In addition to rendering technologies, algorithms are proposed to select optimal numbers of time steps for in situ visualization [19]. GoldRush [35] uses idle resources for in situ processing. Bennett et al. [3] explored a method to offload the computations of merge trees into secondary resources. In our study, we decouple the heavyweight vortex analysis in an independent process for parallel and asynchronous processing, in order to reduce the slowdown of the simulation.

In situ infrastructures are bridges between the simulation code and visualization methods. Typical examples include ParaView Catalyst [11] and VisIt Libsim [30], which can help scientists and visualization practitioners couple the simulation with production visualization tools. Because visualization algorithms are usually I/O intensive, various I/O solutions are proposed for in situ processing, such as ADIOS [17], FlexIO [36], and DataSpaces [7]. Instead of using existing frameworks and I/O solutions, based on the characteristics of both our simulation and analysis algorithms, we customize our in situ workflow and store the output data in high performance databases.

Recently, a group of visualization researchers started the “in situ terminology project” [6], in order to formalize the description of in situ methods with proper terminologies. We follow this project to characterize our methods in several axes. Both integration type and data access are direct, because the primitive detection code is directly plugged into the TDGL simulations and shares the same address space. The proximity is twofold: the primitive detector shares the same GPU cores as the simulation, and the vortex analyzer uses CPUs on or off the node. The synchronization is hybrid: the primitive detector runs synchronously with the simulation, and the vortex analyzer runs asynchronously with the simulation. The operation controls are not applicable in our workflow, and the output type is explorable.

2.3 TDGL simulations

Until recently, TDGL simulations, which are computationally expensive, were limited to 2D [5] or small-scale 3D problems [9]. To study electromagnetic properties of type-II superconductors at larger scales, scientists have developed a TDGL implementation, GLGPU [24]. It is a CUDA-based finite-difference PDE solver that runs on a single node. The size of the model is currently limited by the GPU memory, but the time-varying output can be arbitrarily large.

In our study, we design in situ visualization workflows for GLGPU. The input parameters include the size of the model, configuration of material inclusions, external magnetic field, and external current. At every iteration, the output order parameter field is available on the GPU memory in the format of real and imaginary parts. The simulation also generates other measurements such as voltage—the indicator of energy dissipation of the material. Because the simulation is GPU-based, we use GPU-accelerated algorithms to detect primitives and use idle CPU resources for complex vortex analysis.

3 In situ vortex visualization workflow

Our in situ visualization workflow (Figure 2) consists of two major components: the online processing and post hoc visualization.

The online processing is pipelined with the primitive detector and vortex analyzer, which play the role of a producer and consumer, respectively. The primitive detector, which is tightly-coupled with the simulation process, detects primitives ($\mathcal{F}$ and $\mathcal{E}$) right after every iteration of the simulation. The vortex analyzer, which is loosely-coupled with the simulation, transforms the primitives into vortices, tracks the vortices over time, and analyzes the properties of vortices on the fly. The tracked vortices along with the analysis results are written to the storage for post hoc visualization.

The rationale of the producer-consumer design is threefold: reducing data movement, minimizing the slowdown of the simulation, and avoiding dependencies in the simulation code. First, the producer, or the primitive detector, reduces data movement between GPU and main memory. The size of the primitives is much smaller than that of the order parameter field; thus we copy back the primitives instead of order parameters and conserve scarce bandwidth. Second, the consumer, or the vortex analyzer, minimizes the slowdown of the simulation. At every time step, the simulation continues after an asynchronous request is posted to transmit the primitives. Further analysis will not interfere with the simulation at all. Third, our implementation, especially the vortex analyzer, is independent of the simulation code. Embedding visualization and analysis into simulation code causes extra overhead on management and communication. Our design finds a balance to couple highly optimized kernels with the GPU-based simulation and do the rest of the analysis independently. More details on the primitive detector and the vortex analyzer are in Sections 4 and 5, respectively.

The post hoc visualization enables users to explore and further analyze vortex dynamics that are drawn from the simulation. The tool not only contains traditional 3D visualizations but also provides 2D visualizations of events and distances between vortices, which help users explore when vortices recombine and how they move before and after the events. The web-based tool provides a remote visualization solution and works flexibly on workstations or mobile devices without installing any specific software.

4 Tightly-coupled and GPU-accelerated primitive detection

We develop a GPU-accelerated algorithm for the tightly-coupled vortex primitive detection in TDGL simulations. We also propose a method to use tetrahedral subdivision to eliminate ambiguities of vortex extraction in a hexahedral mesh.

4.1 Mesh subdivision

We detect primitives in tetrahedra instead of hexahedra to prevent ambiguities in vortex extraction and tracking. In previous studies that extract vortices in a hexahedral mesh, ambiguous cases could appear if two vortices penetrates the same hexahedron at the same time. For example, in Figure 3(a), four punctured faces are detected, but we cannot determine their correspondence. Because such ambiguities are guaranteed to not exist in tetrahedral meshes, we subdivide the mesh into tetrahedra and detect primitives in the new mesh. The theoretical foundation is Lemma 2 given in the Appendix.

We subdivide each hexahedron into six tetrahedra, as illustrated in Figures 3(b), (c), and (d). Other subdivision schemes are possible, but our subdivision is conformal, which means that the tetrahedra share edges with neighbor cells. In the subdivided mesh, we index the elements (cells, faces, edges) by the Cartesian coordinates (i, j, k) of the corner node (A in Figure 3(b)) and the type. After removing the duplicated cases, there are 6 types of cells, 12 types of faces, and 7 types of edges in the mesh. Notice that special treatments are necessary for mesh boundaries. The mesh graph that records the cell-face face-edge adjacency can also be built implicitly in further analysis.

4.2 GPU-accelerated primitive detection

We use the GPU to detect primitives in the tetrahedral mesh for vortex extraction. The inputs are the order parameter field of the simulation, and the outputs are the punctured faces and intersected space-time edges for vortex extraction and tracking. Each GPU thread is in charge of a single mesh element (a face or a space-time edge), which can be indexed by a unique ID.

In CUDA, threads are executed in a thread block that typically consists of tens or hundreds of threads. Threads in the same block share a small piece of high speed memory, so-called shared memory. All threads can access the global memory on the GPU, which is much larger but slower than the shared memory. Outside the GPU is the host memory, and the bandwidth between the host memory and the global memory is limited. Our algorithm takes advantage of the GPU thread model and the memory hierarchy for accelerated tightly-coupled primitive detection.

4.2.1 Punctured face detection

The input data of punctured face detection—real and imaginary parts of the order paramter field—are available in the GPU global memory after every iteration of the simulation. We examine every mesh face if punctured by a vortex by computing the phase jump over the mesh boundary by the contour integral defined in Equation 1. If the contour integral is $\pm 2\pi$, the face is punctured; the punctures located where both real and imaginary parts of the order parameter are zero.

In the parallel execution, each GPU thread is in charge of a single mesh face. A list is used to collect detection results. We use a memory buffer in the global memory to store the list. An atomic integer records the end location of the list. To reduce memory contention, for each CUDA thread block, we first store the results in shared memory and then merge them into the list in global memory after all threads are finished in the thread block. The list of punctured faces, each of which consists of the face ID, chirality, and the puncture location, is copied back to the host memory for further analysis.

We must carefully manage the memory footprint of our analysis, because the GPU memory budget is tight in the simulation. First, we use a GPU memory buffer that is allocated by the simulation in a time-sharing way. The GPU memory buffer is not used by the simulation during our analysis and has the same capacity as the order parameter field. In rare cases, such as the very early stage of the simulation that is not of scientific interest, the punctured face list might overflow the buffer, so we have to drop the current timestep in this case. Second, we store the list of punctured faces in a compact way. We use only 12 or 16 bytes for a punctured face, depending on the total number of faces in the mesh. If the mesh has fewer than 2^{31} faces, we use 31 bits for the face ID and 1 bit for chirality, otherwise 63 bits for the face ID and 1 bit for chirality. The 63 bits suffice because there are not likely to be more than 2^{63} faces in the current GLGPU implementation. We store the puncture location—the barycentric coordinates on the face—with two 32-bit floats, so a punctured face takes 12 or 16 bytes in total.

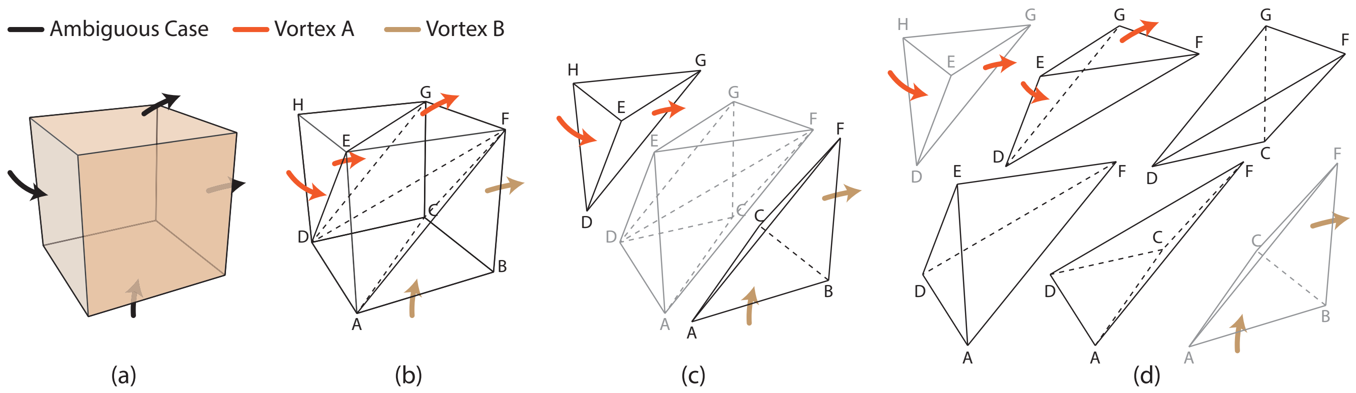


Figure 3: Mesh subdivision for ambiguity-free vortex extraction and tracking: (a) the ambiguity case that a hexahedron intersected by two vortices; (b) the unambiguous results that vortex A punctures cells $DEHG$ and $DEFG$ while vortex B punctures $ABCF$; the punctured faces are DEH, DGE, ABC, and BFC; (c) and (d) show how a hexahedron is subdivided into six tetrahedra.

4.2.2 Intersected space-time edge detection

The input data of the intersected space-time edge detection are the phase field of the two adjacent iterations in the simulation, and the outputs are a list of intersected space-time edges. The detection of space-time edges is similar to that of punctured faces. For each edge in the mesh, we extrude the edge into the time dimension and compute the integral over the space-time contour. The space-time edge is intersected if the contour integral equals $\pm 2\pi$.

Similar to the punctured face detection, each GPU thread is in charge of an edge, and we use the shared memory to avoid contentions in the global memory on the GPU. Each intersected space-time edge is a tuple of the edge ID and chirality, which can be encoded with 4 or 8 bytes depending on the total number of edges in the mesh. Upon completing the detection, the results are transferred back to the host memory. The amount of the output is usually smaller than that of punctured faces. The intersected space-time edge detection takes about 20% of the simulation time, which is less than for the punctured face detection.

Overall, the primitive detection greatly reduces the amount of data for further analysis. Typically, the amount of output is only 10^{-5} to 10^{-2} of the order parameter field, depending on the data complexity. In our benchmark, the GPU-accelerated detection is more than 200 times faster than a CPU-based implementation. We further avoid unnecessary frames by adaptive strategies as described in the in following sections.

4.3 Simulation coupling and adaptive detection

The pseudocode of the simulation coupling is in Algorithm 1. We tightly couple the GPU-accelerated primitive detection with the simulation; thus there is no data movement for the punctured face detection. The intersected edge detection needs to copy the phase field in the previous timestep, but the cost to copy data on the same GPU is minor. The time of copying results back to the main memory is also negligible compared with the detection.

We further reduce the overall online processing time by adapting the primitive detection. If we detect the primitives in every iteration, the primitive detection takes about 50% of the computation time in the simulation. We instead skip a frame if there are no intersected space-time edges. The rationale of the adaptive detection is based on the vortex tracking algorithm. If there are no intersected space-time edges for timesteps $i\prime$ and i ($|\mathcal{E}^i_{i\prime}| = 0$), the IDs of punctured faces remain the same, but the puncture locations may change within the bounds of those faces. We regard the movement of vortices as negligible in this case. Depending on the stability of simulations, we can skip up to 95% of the timesteps in our experiments.

Algorithm 1 Main loop of the tightly-coupled vortex primitive detection. `TDGL` is the simulation; `comm` is the communicator, and `isend()` is the non-blocking send. $\psi\prime$ and ψ are the order parameter fields of the previous and the current time step, respectively; $i\prime$ and i are the time step of the previous and the current time step, respectively. PFs and IEs are abbreviations for punctured faces and intersected space-time edges, respectively.

```
$i \leftarrow 0$
while !TDGL.finished() do
    $\psi \leftarrow$TDGL.iterate()                    ▷ Get $\psi$ from the simulation
    if $i = 0$ then                                     ▷ Process the first time step
        $\mathcal{F}_i \leftarrow$detect_PFs_GPU($\psi$)             ▷ Detect PFs
        comm.isend(tag='F', $\{i, \mathcal{F}_i\}$)           ▷ Send PFs
    else
        $\mathcal{E}^i_{i\prime} \leftarrow$detect_IEs_GPU($\psi\prime$, $\psi$)         ▷ Detect IEs
        if $|\mathcal{E}^i_{i\prime}| = 0$ then
            continue   ▷ If vortices do not move, skip the current time step
        else
            comm.isend(tag='E', $\{i\prime, i, \mathcal{E}^i_{i\prime}\}$) ▷ Send IEs
            $\mathcal{F}_i \leftarrow$detect_PFs_GPU($\psi$)
            comm.isend(tag='F', $\{i, \mathcal{F}_i\}$)
            $\{i\prime, \psi\prime\} \leftarrow \{i, \psi\}$        ▷ Copy the current time step
        end if
    end if
    $i \leftarrow i + 1$
end while
```

5 Loosely-Coupled and Task-Parallel Vortex Analysis

The vortex analyzer extracts, tracks, analyzes, and reduces vortices in the in situ workflow. The main challenge in the vortex analyzer is the unbalanced producer (simulation/primitive detector) and consumer (vortex analyzer) rate. Hence, we design and implement a task-parallel graph analysis scheme. Because the producer can generate vortex primitives much faster than the consumer can, the consumer could overflow memory. We schedule the graph analysis tasks on different threads in parallel, in order to improve the consuming rate.

5.1 Parallel task execution

As shown in Figure 4, our task-parallel model is an acyclic directed graph, where the nodes are data and tasks and the links define their dependencies. There are two types of data nodes: punctured faces $\mathcal{F}_i$ and intersected space-time edges $\mathcal{E}^i_{i\prime}$, where $i\prime$ and i are the in-

Algorithm 2 Main loop of the loosely-coupled vortex extraction and tracking. `sched` is the scheduler of the task-parallel execution with dependencies, and `comm` is the communicator. `Ex` and `Tr` are extraction and tracking tasks, respectively.

```
sched.start(max_num_threads)
while comm.probe() and not sched.overflowed() do
▷ Test if message comes in and the scheduler is not overflowed
    comm.recv(&tag, &data)
    if tag=`F' then
        $\{i, \mathcal{F}_i\} \leftarrow$data
        sched.add(Ex$_i$($\mathcal{F}_i$), depend=$\phi$)
    else if tag=`E' then
        $\{i\prime, i, \mathcal{E}^i_{i\prime}\} \leftarrow$data
        sched.add(Tr$^i_{i\prime}$($\mathcal{E}^i_{i\prime}$)), depend=$\{$Ex$_{i\prime}$, Ex$_i\}$)
    end if
end while
sched.wait_for_all()                    ▷ Wait until all tasks finish
```

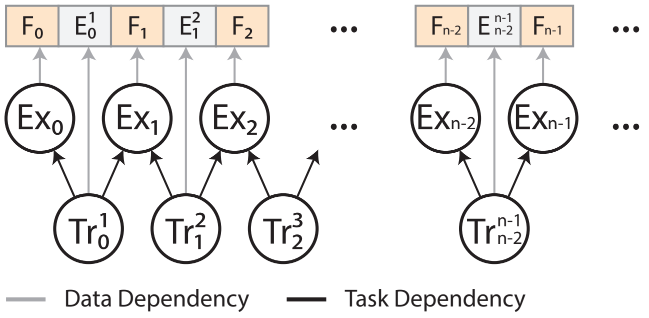


Figure 4: Dependency graph in the parallel execution of vortex extraction and tracking. Each extraction task $\mathtt{Ex}_i$ depends on the punctured faces $\mathcal{F}_i$; each tracking task $Tr^i_{i\prime}$ depends on the intersected space-time edges $\mathcal{E}^i_{i\prime}$ and the extraction tasks $\mathtt{Ex}_{i\prime}$ and $\mathtt{Ex}_i$.

dices of timesteps. Likewise, there are two types of tasks: extraction tasks ($\mathtt{Ex}_i$) and tracking tasks ($\mathtt{Tr}^i_{i\prime}$). During the run, the data nodes are streamed into the graph, and the tasks are dynamically created. The extraction task $\mathtt{Ex}_i$ depends on the punctured face list $\mathcal{F}_i$. The tracking task $\mathtt{Tr}^i_{i\prime}$ depends on multiple nodes, including $\mathcal{F}_{i\prime}$, $\mathcal{F}_i$, and $\mathtt{Tr}^i_{i\prime}$.

We use the thread pool model to execute the tasks in parallel. The model consists of the main thread and a number of worker threads. The main thread manages the communication and task scheduling, and the worker threads execute tasks. The pseudo code of the main thread loop is shown in Algorithm 2. Notice that we must control the task scheduler to prevent the producer from overflowing the consumer. If memory is insufficient to schedule an upcoming task, we have to block communication and wait until some tasks finish and release enough memory. The blocking of communication may eventually block the simulation in this case.

5.2 Online vortex analysis and reduction

We analyze and reduce the vortices on the fly as soon as they are extracted and tracked. Based on requirements from the scientists, we generate the following analysis results that represent the dynamics of vortices. The outputs are also stored in a database with the vortices.

Events. The vortex events are defined as topological changes of vortices: birth, death, split, merge, and recombination. The event detection is based on the association graph. For example, recombination happens if two vortices in the current timestep are associated with another two vortices in the next timestep. The in situ detection of events can accurately tell when and which vortices are involved

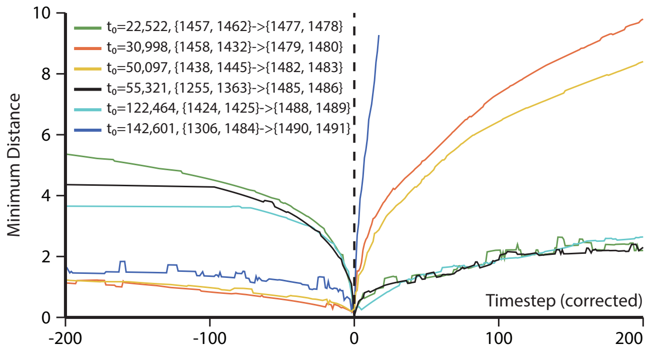


Figure 5: The distance plot (minimum distance between vortices before and after recombination) in the `Crossing` simulation. The legends show the timestep of recombination and the involved vortices of the event. The dark blue line corresponds to the event in Figure 10.

in the event.

Distance. We compute the minimum distance between vortices at every time step. The output of the distances is stored in a matrix. Scientists are interested in how distance between vortices changes over time, especially for vortices that recombine. In the following sections we also use the matrix for distance projection in order to provide an occlusion-free 2D visualization.

Moving speed. The moving speed V of a vortex, which is directly related to the energy dissipation of superconducting material, is defined as the area swept by the vortex in unit time. The swept area of an interval is computed by first tessellating the vortices in the two time steps into triangles and then summing up the areas of the triangles. The results are stored with the output vortices.

The data reduction of vortices is based on the work of Phillips et al. [22], which reduces the data size by geometry simplification and parameterization. First, the vortices are simplified with the Ramer-Douglas-Peucker [8] algorithm, which approximates an input curve by a series of points. Second, we use Bezier curves to dynamically fit the simplified curve [25]. Both data reduction algorithms are lossy but error bounded. Notice that errors in the former algorithm are measured by the maximum distance between the simplified curve and the input curve, and the errors in the latter algorithm are measured by the squared error between the input and fitted curve. Given a reasonable error bound (0.1 and 0.01 in our experiments), the output vortices can reduce up to 90% in storage size.

6 Post hoc visualization

We provide post hoc visualization for users to explore the online analysis results. The visualization involves four linked views: spatial, timeline, distance projection, and distance. The design goals are to help scientists explore spatial distributions of vortices and analyze the changes of vortices over time.

The spatial view visualizes the 3D geometries of vortices and material inclusions in the simulation. Users can identify and explore important spatial structures of vortices. In this view, the vortex curves are extruded into tubes for better shading and perception than possible with plain line geometries. The color indicates the ID of vortices. The color of a vortex remains the same over time unless the vortex has a topological change.

The timeline view visualizes the events over time with glyphs. Users can navigate the time-varying vortices by moving the cursor on the timeline view.

The distance view (Figure 5) visualizes how distances between vortices change before and after recombinations. The x-axis is the relative time with respect to the time of recombination, and the y-axis is the minimum distance of vortices that cross with each other

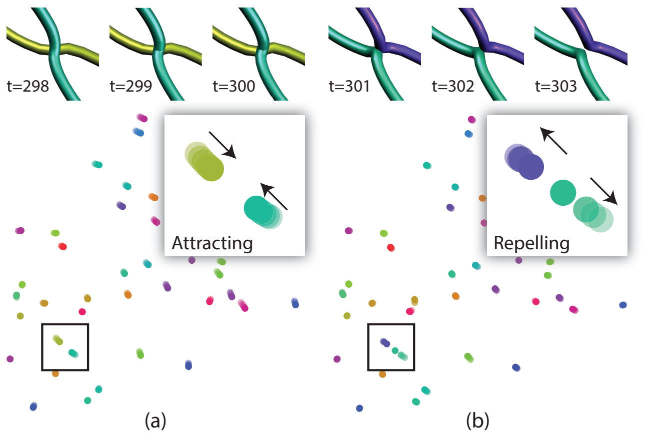


Figure 6: Distance projection in the `Unstable_BX` simulation. Two vortices approach (a) and then repel (b) each other before and after the recombination. The opacity in the figure encodes the time.

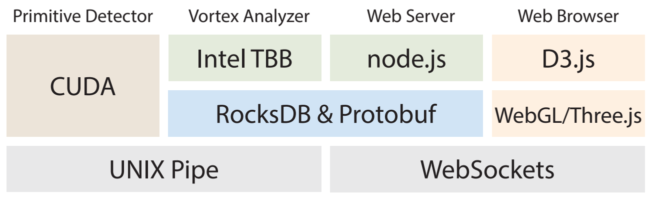


Figure 7: Software stack of our in situ vortex visualization framework. Various tools and APIs are used in both online processing and post hoc visualization.

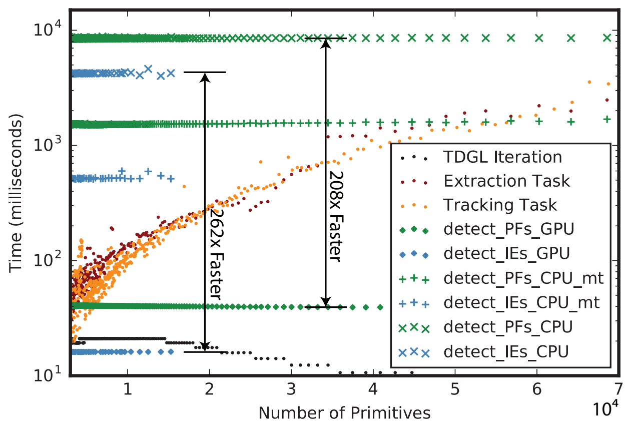


Figure 8: Timings of TDGL iterations (`Bramp` simulation), primitive detection, vortex extraction, and vortex tracking with respect to different numbers of primitives. Timings of `detect_*_GPU`, `detect_*_CPU`, and `detect_*_CPU_mt` correspond to the primitive detection with GPU, single-threaded CPU, and multithreaded CPU, respectively.

at time 0. We design this view because scientists would like to know how the minimum distance changes before and after a recombination. Users can also select vortices to see how they morph over time, which is the key in the study of vortex dynamics.

The distance projection view (Figure 6) maps the minimum distances between vortex lines into 2D space for individual time steps. We use the multidimensional scaling algorithm to transform the input distance matrix into a set of points in 2D space. The distance between points in the 2D space is preserved as much as possible. The rationales for the projection view is twofold. First, the projection view provides a 2D visualization of vortices without any occlusions. Although the geometric information in the projection is lost, the distance information is still kept in the 2D projections. Second, the projection view can visualize the change of distances between time steps by animation. The projection provides a visual clue about the distance changes and helps users select vortices of interest.

7 Implementation

We use various tools and APIs to build our in situ visualization workflow. The software stack of our implementation is illustrated in Figure 7.

The tightly-coupled primitive detector is implemented with CUDA and C, without any third-party libraries. The output primitives are sent to the loosely-coupled vortex analyzer asynchronously via UNIX pipes. The primitives can also be transferred to the vortex analyzer running on a remote machine by redirecting the pipe with the `netcat` command in UNIX.

The loosely-coupled vortex analyzer is written in C++11. The Intel Threading Building Blocks (TBB)[1] library provides APIs to schedule the task-parallel execution and lock-free data structures to achieve high concurrency. The output data, including vortex lines, association matrices, events, and all other analysis results, are serialized with Google protocol buffers[2] and then written to storage by Facebook RocksDB.[3] We store data in a database instead of files because RocksDB provides a high performance persistent key-value store optimized for multithreaded writes. We use `bzip2` to further compress the output in the database. Notice that the vortex analyzer can run on either the same node as the simulation or a standalone node with network connections. We usually run both the primitive detector and the vortex analyzer on the same node to fully utilize the hardware resources of the GPU and all CPU cores.

The web server is hosted with Node.js,[4] which is a server-side JavaScript runtime. We build bindings of our C++11 code for Node.js, in order to enable access to the output data in the database. The web server and clients exchange queries and results via WebSockets, which provides full-duplex communication.

The web client is implemented with WebGL and D3.js [4]. The 3D interactive visualization is based on Three.js,[5] which manages the scene graph and wraps WebGL. The 2D visualizations are rendered in SVG and overlaid on top of the 3D view.

8 Performance Benchmark

The performance benchmarks are shown in Table 1, Figure 8, and Figure 9. The benchmark was conducted on a workstation equipped with two quad-core Intel Xeon E5620 CPUs, 12 GB main memory, and an NVidia K40c GPU. Each CPU supports up to 8 hardware threads, and the GPU has 2,880 stream cores and 12 GB memory. The simulation process that manages the GPU simulation and primitive detection is single threaded, and the vortex tracker can use up to all eight CPU cores on the machine. The output data are written to the database on a local hard disk drive. We analyze the performance of our in situ workflow in four aspects: speedup of the GPU-accelerated primitive detection, efficiency of adaptive primitive detection, performance of task-parallel vortex analyzer, and I/O cost.

First, we measure the speedup of the GPU-accelerated primitive detection algorithms in Section 4. The timings with respect to the different numbers of primitives are shown in Figure 8. The baseline

[1] `https://www.threadingbuildingblocks.org/`

[2] `https://developers.google.com/protocol-buffers/`
[3] `http://rocksdb.org/`
[4] `https://nodejs.org/`
[5] `http://threejs.org/`

Table 1: Simulation specifications and timings. F_{all} is the total number of timesteps in the simulation, and $F_{skipped}$ is the number of skipped timesteps in the adaptive primitive detection. T_{total}, T_f, and T_e are the total simulation time, punctured face detection time, and intersected edge detection time in seconds, respectively. $T^{(async)}_{analysis}$ is the CPU time of the vortex analyzer, which does not affect T_{total} because the analysis is asynchronous. S_{sim}, S_p, and S_{out} are the size of order parameter field data, vortex primitives, and final outputs in the database, respectively.

Name	Resolution	F_{all}	$F_{skipped}$	T_{total}	T_f	T_e	$T^{(async)}_{analysis}$	S_{sim}	S_p	S_{out}	S_{out}/S_{sim}
`Bramp`	$128 \times 512 \times 64$	63,000	59,614	1,550.92	138.07	1,013.16	340.93	1.923 TB	67 MB	16 MB	0.0008%
`Unstable_BX`	$256 \times 128 \times 32$	200,000	1,211	3,492.96	2,011.55	801.50	4,196.25	1.53 TB	9.53 GB	477 MB	0.03%
`Crossing`	$256 \times 256 \times 128$	830,000	261,829	82,527.33	45,803.19	26,874.04	73,250.32	4.75 TB	103 GB	11 GB	0.23%

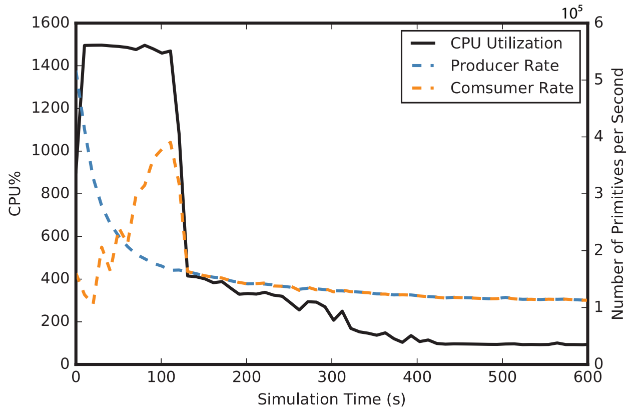


Figure 9: CPU usage and the producing/consuming rate in the task-parallel execution (`Bramp` simulation).

approach is a software implementation that can run on either single or multiple (eight in the experiments) CPU threads. The timings of GPU algorithms include the data transfer, that is, the copy back of primitives from the GPU memory. The performance of the primitive detection does not change significantly with the number of primitives. On average, the punctured face detection is $208\times$ and $37\times$, compared with the single-threaded and multithreaded CPU implementation, respectively. Likewise, the intersected space-time edge detection on the GPU is $262\times$ and $32\times$ faster than the single-threaded and multithreaded CPU code, respectively.

Second, we evaluate the efficiency of the adaptive primitive detection strategy in Section 4.3. The criterion to skip a time step, namely, the number of intersected space-time edges, is zero, because they imply the movement of vortices and we ignore a time step that has minimal change to the previous one. As shown in Table 1, the percentage of skipped time steps ranges from 1% to 95%; thus the efficiency highly depends on the simulation. In general, if the simulation is more stable, the adaptive strategy can save more time in the in situ workflows.

Third, we benchmark the loosely coupled, asynchronous, and task-parallel vortex tracker in Section 5. Figure 8 shows the timings of extraction tasks and tracking tasks with respect to different numbers of primitives. We can see that the execution time of an individual task grows linearly with the number of primitives. In most cases, the execution time is much longer than that of the producer (simulation and primitive detection), which confirms that we must use parallelism to boost the performance of the consumer (vortex detector). Figure 9 visualizes the CPU utilization of the in situ workflow as the simulation time elapses. The CPU utilization reflects how many CPU cores are used. Notice that the online data analysis and I/O also run in the experiment. The CPU utilization of the producer remains almost constant (100%), because the GPU is managed by a single CPU thread. The CPU utilization of the consumer, which is proportional to the number of primitives, varies from 10% to 1500%. At the early stage of the simulation, the system is highly unstable; thus there are usually more vortices, and the workload of the vortex tracker is higher. After the simulation is stabilized, the workload is also stabilized—the workload of the consumer is about three times higher than that of the producer.

Fourth, we measure the I/O cost of the in situ workflow. The output data written in the database is only about 10^{-5} to 10^{-2} of the order parameter field data in the simulation. As recorded by the database log, the peak output I/O bandwidth is about 30 MB/s, which is much more affordable than storing the raw simulation outputs.

Overall, the cost of the whole in situ workflow $((T_f + T_e)/T_{total})$ is 75%$\sim$88% of the simulation, depending on the data complexity. Notice that T_f and T_e are already much less than they used to be with CPUs, and we also hide $T^{(async)}_{analysis}$ in T_total by the asynchronous task-parallel execution. The in situ workflow also greatly reduces the amount of output while obtaining the most precise analysis results.

9 Application Results

We demonstrate application cases of in situ vortex visualization. The specifications of the simulations are in Table 1.

9.1 `Unstable_BX` simulation

The `Unstable_BX` simulation is conducted in order to understand the periodic energy dissipative states in a superconductor with several inclusions. The magnetic field is aligned with the external current, which would result in no vortex dynamics (due to the absence of Lorentz forces). However, thermal fluctuations and material defects, modeled as spherical inclusions, lead to interesting behavior. Scientists would like to study when, and how vortices recombine with each other while performing periodic motion.

Our in situ workflow enables the detailed visualization and analysis of vortex dynamics. Figure 1 compares the visualization results from both traditional post hoc methods and in situ methods. Previously, scientists saved the order parameter field for every 100 timesteps and then analyzed the data in low temporal resolution after the simulation. As shown in Figures 1(c) and (d), a "compound" event happens, which involves vortices #588, #489, and #485 at timestep 6,900 and vortices #590, #593, #592 at timestep 7,000. However, we cannot read any more details within the 100 timesteps. The in situ processing enables us to see two recombination events. At timestep 6990, two vortices #588 and #489 cross each other and swap parts. Two new vortices #590 and #591 are generated after the recombination at timestep 6991. Right after this event, #590 and #485 recombine into #592 and #593. Another example is shown in Figure 6, which visualizes a recombination event with distance projection. We can observe how vortices deform, bend, attract each other, and then repel each other after the events.

9.2 `Crossing` simulation

Scientists study vortex crossings in superconducting slabs that are induced by tilting the applied magnetic field [29]. The way vortices in superconductors cross each other has been studied for many years, but most studies were performed either under artificial conditions or on very small lengths scales. The GLGPU simulation makes it possible to reveal realistic situations. The in situ visualization of vortices and their distances gives a good impression of the

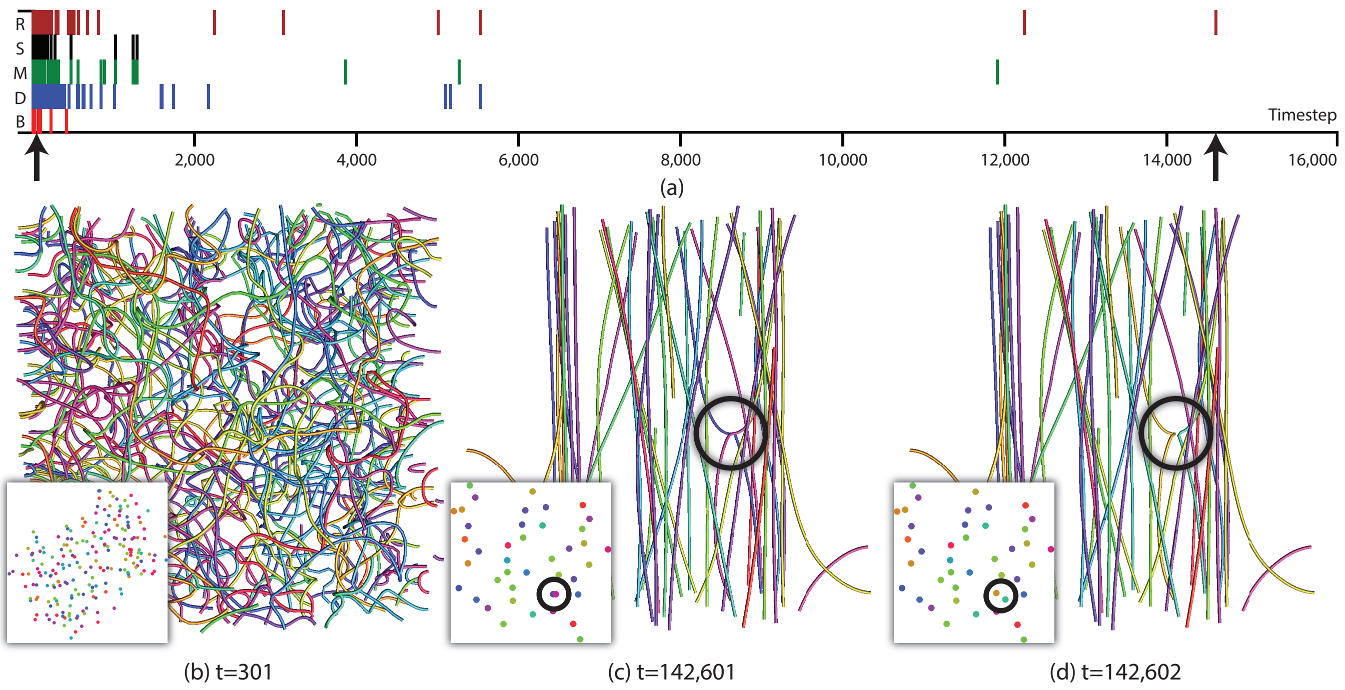


Figure 10: In situ vortex visualizations of the `Crossing` simulation: (a) the timeline view showing different types of events (B: birth, D: death, M: merge, S: split, R: recombination) over time; (b) the early stage of the simulation; (c) and (d) a recombination event that is highlighted by the black circles. The distance projection plots are in the left bottom corner of the images.

physics of these crossing events, which typically consist of merge, split, and recombination.

As shown in Figure 10, the early stage of the simulation is unstable and chaotic. Vortices are mostly stable in the later stage, yet a few recombinations happen. A traditional post hoc workflow is difficult to capture the rapidly changing vortices near recombinations. Now we can find the exact time of the events and observe how vortices morph before and after the events.

We generate the distance plot (Figure 5) in situ, recently was used to study various vortex recombinations scenarios [12]. Previously in the post hoc workflow, scientists had to run the simulation twice to generate the distance plot. In the first "full" run, the order parameter field is stored every 250 iterations. After the first run, scientists choose an interval of interest that involves a recombination event. In the second "detail" run, every iteration in this interval (2,000 timesteps) is stored for post hoc analysis. Such a workflow requires a manual process and computation resources, and it may miss important features in the simulation.

With the in situ workflow, the distance plot can be automatically generated with no manual operations or additional simulation runs. The recombination events and vortex curves are extracted and stored based on every single iteration in situ. Compared with the traditional post hoc workflow, the in situ workflow enables scientists to visualize and analyze the vortex dynamics in greater detail and with less work than previously possible.

10 Conclusions and Future Work

In this paper, we present an in situ vortex visualization framework for GPU-based TDGL simulations. The system consists of tightly-coupled GPU-accelerated primitive detection for ambiguity-free vortex analysis, loosely-coupled task-parallel feature tracking, and web-based visualizations. Applications show that the only solution to our driving scientific problem—finding vortex recombinations in the rapidly changing simulations—is the in situ workflow.

We would like to extend our study in four directions. First, we are going to design the in situ workflow for the next-generation TDGL simulation, which is based on an unstructured mesh in distributed memory. Simulation steering is also going to be a necessary component in the future system. Second, we would like to apply our task-parallel design to other feature tracking problems. Third, we are going to develop a scalable solution to visualize events in long time sequences for in situ visualization. Fourth, we plan to use our in situ workflow in other complex-valued Ginzburg-Landau simulations such as Bose-Einstein condensation and superfluidity.

Appendix

In this appendix, we prove that a tetrahedron in our mesh subdivision cannot be punctured by two vortices and therefore ambiguity-free vortex extraction is guaranteed. In the discrete ψ field, we calculate the phase jump over a mesh edge AB by

$$\Delta_{AB} = \int_{AB} \nabla\theta \cdot d\mathbf{l} = \mod(\theta_B - \theta_A + \pi, 2\pi) - \pi, \quad (2)$$

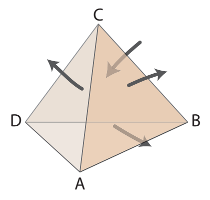
where θ_A and θ_B are the phase angle on nodes A and B, respectively. Notice that the modulo maps Δ_{AB} into the range of $[-\pi, \pi)$, so we cannot measure any larger phase jumps over mesh edges than this range.

Lemma 1. *If $x + y + z = 2\pi$, $x, y, z \in [-\pi, \pi)$ then $x, y, z > 0$.*

Proof. Assume $z \leq 0$. Then $x + y \geq 2\pi$. A contradiction occurs because $x < \pi$ and $y < \pi$. □

Lemma 2. *There is at most one pair of punctured points on a tetrahedron, if the phase jump over every edge of this tetrahedron is in the range of $[-\pi, \pi)$.*

As shown in the right figure, in tetrahedron $ABCD$, the phase jumps over every edge Δ_{AB}, Δ_{BC}, Δ_{CA}, Δ_{CD}, Δ_{DA}, and Δ_{BD} are in the range of $[-\pi, \pi)$. We prove this lemma by contradiction.



Proof. Without loss of generality, we assume there are two pairs of punctured faces, so that $\Delta_{AB} + \Delta_{BC} + \Delta_{CA} = 2\pi$, $\Delta_{AC} + \Delta_{CD} + \Delta_{DA} = 2\pi$, $\Delta_{AD} + \Delta_{DB} + \Delta_{BA} = -2\pi$, and $\Delta_{BD} + \Delta_{DC} + \Delta_{CB} = -2\pi$. Based on Lemma 1, we have $\Delta_{CA} > 0$ and $\Delta_{AC} > 0$. A contradiction occurs because $\Delta_{CA} = -\Delta_{AC}$. The lemma is true because relabeling the tetrahedron can cover all cases (any pair of faces have common edge, which is AC in our case). $\square$

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Science, Division of Materials Science and Engineering.

References

[1] J. P. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *SC'14: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434, 2014.

[2] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In situ methods, infrastructures, and applications on high performance computing platforms. *Comput. Graph. Forum*, 35(3):577–597, 2016.

[3] J. Bennett, H. Abbasi, P.-T. Bremer, R. W. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. P. Pébay, D. C. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC'12: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 49:1–49:9, 2012.

[4] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011.

[5] X. H. Chao, B. Y. Zhu, A. V. Silhanek, and V. V. Moshchalkov. Current-induced giant vortex and asymmetric vortex confinement in microstructured superconductors. *Physics Review B*, 80(054506):1–6, 2009.

[6] H. Childs et al. The in situ terminology project. `https://ix.cs.uoregon.edu/~hank/insituterminology/`.

[7] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[8] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[9] Q. Du. Numerical approximations of the Ginzburg-Landau models for superconductivity. *Journal of Mathematical Physics*, 46(095109):1–22, 2005.

[10] N. Fabian. In situ fragment detection at scale. In *LDAV'12: Proc. IEEE Symposium on Large Data Analysis and Visualization*, pages 105–108, 2012.

[11] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. E. Rasquin, and K. E. Jansen. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. In *LDAV'11: Proc. IEEE Symposium on Large Data Analysis and Visualization*, pages 89–96, 2011.

[12] A. Glatz, V. K. Vlasko-Vlasov, W. K. Kwok, and G. W. Crabtree. Vortex cutting in superconductors. *Physical Review B*, 94(064505):1–11, 2016.

[13] H. Guo, C. L. Phillps, T. Peterka, D. Karpeyev, and A. Glatz. Extracting, tracking and visualizing magnetic flux vortices in 3D complex-valued superconductor simulation data. *IEEE Trans. Vis. Comput. Graph.*, 22(1):827–836, 2016.

[14] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[15] M. Jiang, R. Machiraju, and D. Thompson. Detection and visualization of vortices. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*. Elsevier, 2005.

[16] A. G. Landge, V. Pascucci, A. Gyulassy, J. Bennett, H. Kolla, J. Chen, and P. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC'14: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1020–1031, 2014.

[17] J. F. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IPDPS'09: Proc. IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

[18] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994.

[19] P. Malakar, V. Vishwanath, T. Munson, C. Knight, M. Hereld, S. Leyffer, and M. E. Papka. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *SC'15: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 52:1–52:11, 2015.

[20] R. Peikert and M. Roth. The "parallel vectors" operator–a vector field visualization primitive. In *Proc. of IEEE Visualization '99*, pages 263–270, 1999.

[21] C. L. Phillips, H. Guo, T. Peterka, D. Karpeyev, and A. Glatz. Tracking vortices in superconductors: Extracting singularities from a discretized complex scalar field evolving in time. *Physical Review E*, 93(023305):1–13, 2016.

[22] C. L. Phillips, T. Peterka, D. Karpeyev, and A. Glatz. Detecting vortices in superconductors: Extracting one-dimensional topological singularities from a discretized complex scalar field. *Physics Review E*, 91(023311):1–12, 2015.

[23] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Comput. Graph. Forum*, 22(4):1–17, 2003.

[24] I. Sadovskyy, A. Koshelev, C. Phillips, D. Karpeyev, and A. Glatz. Stable large-scale solver for Ginzburg-Landau equations for superconductors. *Journal of Computational Physics*, 294(C):639–654, 2015.

[25] P. J. Schneider. Graphics gems. chapter An Algorithm for Automatically Fitting Digitized Curves, pages 612–626. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[26] S. Stegmaier and T. Ertl. A graphics hardware-based vortex detection and visualization system. In *Proc. IEEE Visualization '04*, pages 195–202, 2004.

[27] H. Theisel and H.-P. Seidel. Feature flow fields. In *VisSym'03: Proc. Symp. Data Visualization*, pages 141–148, 2003.

[28] A. Tikhonova, C. Correa, and K.-L. Ma. Explorable images for visualizing volume data. In *Proc. IEEE Pacific Visualization Symposium 2010*, pages 177–184, 2010.

[29] V. Vlasko-Vlasov, A. Koshelev, A. Glatz, C. Phillips, U. Welp, and W. Kwok. Flux cutting in high-T_c superconductors. *Physical Review B*, 91(014516):1–16, Jan 2015.

[30] B. Whitlock, J. M. Favre, and J. S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *EGPGV'11: Proc. Eurographics Symposium on Parallel Graphics and Visualization*, pages 101–109, 2011.

[31] J. Woodring, M. Petersen, A. Schmeiber, J. Patchett, J. P. Ahrens, and H. Hagen. In situ eddy analysis in a high-resolution ocean climate model. *IEEE Trans. Vis. Comput. Graph.*, 22(1):857–866, 2016.

[32] Y. Ye, R. Miller, and K.-L. Ma. In situ pathtube visualization with explorable images. In *EGPGV'13: Proc. Eurographics Symposium on Parallel Graphics and Visualization*, pages 9–16, 2013.

[33] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. Ma. In situ visualization for large-scale combustion simulations. *IEEE Comput. Graph. Appl.*, 30(3):45–57, 2010.

[34] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *SC'08: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 48:1–48:11, 2008.

[35] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *SC'13: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 78:1–78:12, 2013.

[36] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. FlexIO: I/O middleware for location-flexible scientific data analytics. In *IPDPS'13: Proc. IEEE International Symposium on Parallel and Distributed Processing*, pages 320–331, 2013.